Appendix

A

Answers to Review Questions

Chapter 1: Java Building Blocks

1. A, B, E. Option A is valid because you can use the dollar sign in identifiers. Option B is valid because you can use an underscore in identifiers. Option C is not a valid identifier because `true` is a Java reserved word. Option D is not valid because the dot (.) is not allowed in identifiers. Option E is valid because Java is case sensitive, so `Public` is not a reserved word and therefore a valid identifier. Option F is not valid because the first character is not a letter, $, or _.
2. D. Boolean fields initialize to `false` and references initialize to `null`, so `empty` is `false` and `brand` is `null`. `Brand = null` is output.
3. B, D, E. Option A (line 4) compiles because `short` is an integral type. Option B (line 5) generates a compiler error because `int` is an integral type, but 5.6 is a floating-point type. Option C (line 6) compiles because it is assigned a `String`. Options D and E (lines 7 and 8) do not compile because `short` and `int` are primitives. Primitives do not allow methods to be called on them. Option F (line 9) compiles because `length()` is defined on `String`.
4. A, B. Adding the variable at line 2 makes `result` an instance variable. Since instance variables are in scope for the entire life of the object, option A is correct. Option B is correct because adding the variable at line 4 makes `result` a local variable with a scope of the whole method. Adding the variable at line 6 makes `result` a local variable with a scope of lines 6–7. Since it is out of scope on line 8, the `println` does not compile and option C is incorrect. Adding the variable at line 9 makes `result` a local variable with a scope of lines 9 and 10. Since line 8 is before the declaration, it does not compile and option D is incorrect. Finally, option E is incorrect because the code can be made to compile.
5. C, D. Option C is correct because it imports `Jelly` by classname. Option D is correct because it imports all the classes in the `jellies` package, which includes `Jelly`. Option A is incorrect because it only imports classes in the `aquarium` package—`Tank` in this case—and not those in lower-level packages. Option B is incorrect because you cannot use wildcards anyplace other than the end of an import statement. Option E is incorrect because you cannot import parts of a class with a regular import statement. Option F is incorrect because options C and D do make the code compile.
6. E. The first two imports can be removed because `java.lang` is automatically imported. The second two imports can be removed because `Tank` and `Water` are in the same package, making the correct answer E. If `Tank` and `Water` were in different packages, one of these two imports could be removed. In that case, the answer would be option D.
7. A, B, C. Option A is correct because it imports all the classes in the `aquarium` package including `aquarium.Water`. Options B and C are correct because they import `Water` by classname. Since importing by classname takes precedence over wildcards, these compile. Option D is incorrect because Java doesn't know which of the two wildcard `Water`

classes to use. Option E is incorrect because you cannot specify the same classname in two imports.

8. B. Option B is correct because arrays start counting from zero and strings with spaces must be in quotes. Option A is incorrect because it outputs `Blue`. C is incorrect because it outputs `Jay`. Option D is incorrect because it outputs `Sparrow`. Options E and F are incorrect because they output `Error: Could not find or load main class BirdDisplay.class`.

9. A, C, D, E. Option A is correct because it is the traditional `main()` method signature and variables may begin with underscores. Options C and D are correct because the array operator may appear after the variable name. Option E is correct because varargs are allowed in place of an array. Option B is incorrect because variables are not allowed to begin with a digit. Option F is incorrect because the argument must be an array or varargs. Option F is a perfectly good method. However, it is not one that can be run from the command line because it has the wrong parameter type.

10. E. Option E is the canonical `main()` method signature. You need to memorize it. Option A is incorrect because the `main()` method must be public. Options B and F are incorrect because the `main()` method must have a `void` return type. Option C is incorrect because the `main()` method must be static. Option D is incorrect because the `main()` method must be named `main`.

11. C, D. Option C is correct because all non-primitive values default to `null`. Option D is correct because float and double primitives default to `0.0`. Options B and E are incorrect because `int` primitives default to `0`.

12. G. Option G is correct because local variables do not get assigned default values. The code fails to compile if a local variable is not explicitly initialized. If this question were about instance variables, options D and F would be correct. A `boolean` primitive defaults to `false` and a `float` primitive defaults to `0.0`.

13. A, D. Options A and D are correct because `boolean` primitives default to `false` and `int` primitives default to `0`.

14. D. The package name represents any folders underneath the current path, which is `named.A` in this case. Option B is incorrect because package names are case sensitive, just like variable names and other identifiers.

15. A, E. Underscores are allowed as long as they are directly between two other digits. This means options A and E are correct. Options B and C are incorrect because the underscore is adjacent to the decimal point. Option D is incorrect because the underscore is the last character.

16. B, C, D. `0b` is the prefix for a binary value and is correct. `0x` is the prefix for a hexadecimal value. This value can be assigned to many primitive types, including `int` and `double`, making options C and D correct. Option A is incorrect because `9L` is a long value. `long amount = 9L` would be allowed. Option E is incorrect because the underscore is immediately before the decimal. Option F is incorrect because the underscore is the very last character.

17. A, E. `Bunny` is a class, which can be seen from the declaration: public class `Bunny`. `bun` is a reference to an object. `main()` is a method.

18. C, D, E. `package` and `import` are both optional. If both are present, the order must be `package`, then `import`, then `class`. Option A is incorrect because `class` is before `package` and `import`. Option B is incorrect because `import` is before `package`. Option F is incorrect because `class` is before `package`. Option G is incorrect because `class` is before `import`.

19. B, D. The `Rabbit` object from line 3 has two references to it: `one` and `three`. The references are nulled out on lines 6 and 8, respectively. Option B is correct because this makes the object eligible for garbage collection after line 8. Line 7 sets the reference `four` to the now null `one`, which means it has no effect on garbage collection. The `Rabbit` object from line 4 only has a single reference to it: `two`. Option D is correct because this single reference becomes `null` on line 9. The `Rabbit` object declared on line 10 becomes eligible for garbage collection at the end of the method on line 12. Calling `System.gc()` has no effect on eligibility for garbage collection.

20. B, E. Calling `System.gc()` suggests that Java might wish to run the garbage collector. Java is free to ignore the request, making option E correct. `finalize()` runs if an object attempts to be garbage collected, making option B correct.

21. A. While the code on line 3 does compile, it is not a constructor because it has a return type. It is a method that happens to have the same name as the class. When the code runs, the default constructor is called and count has the default value (0) for an `int`.

22. B, E. C++ has operator overloading and pointers. Java made a point of not having either. Java does have references to objects, but these are pointing to an object that can move around in memory. Option B is correct because Java is platform independent. Option E is correct because Java is object oriented. While it does support some parts of functional programming, these occur within a class.

23. C, D. Java puts source code in `.java` files and bytecode in `.class` files. It does not use a `.bytecode` file. When running a Java program, you pass just the name of the class without the `.class` extension.

Chapter 2: Operators and Statements

1. A, D. Option A is the equality operator and can be used on numeric primitives, `boolean` values, and object references. Options B and C are both arithmetic operators and cannot be applied to a `boolean` value. Option D is the logical complement operator and is used exclusively with `boolean` values. Option E is the modulus operator, which can only be used with numeric primitives. Finally, option F is a relational operator that compares the values of two numbers.

2. A, B, D. The value `x + y` is automatically promoted to `int`, so `int` and data types that can be promoted automatically from `int` will work. Options A, B, D are such data types. Option C will not work because `boolean` is not a numeric data type. Options E and F will not work without an explicit cast to a smaller data type.
3. F. In this example, the ternary operator has two expressions, one of them a `String` and the other a `boolean` value. The ternary operator is permitted to have expressions that don't have matching types, but the key here is the assignment to the `String` reference. The compiler knows how to assign the first expression value as a `String`, but the second `boolean` expression cannot be set as a `String`; therefore, this line will not compile.
4. B, C, D, F. The code will not compile as is, so option A is not correct. The value `2 * x` is automatically promoted to `long` and cannot be automatically stored in `y`, which is in an `int` value. Options B, C, and D solve this problem by reducing the `long` value to `int`. Option E does not solve the problem and actually makes it worse by attempting to place the value in a smaller data type. Option F solves the problem by increasing the data type of the assignment so that `long` is allowed.
5. C. This code does not contain any compilation errors or an infinite loop, so options D, E, and F are incorrect. The `break` statement on line 8 causes the loop to execute once and finish, so option C is the correct answer.
6. F. The code does not compile because two `else` statements cannot be chained together without additional `if-then` statements, so the correct answer is option F. Option E is incorrect as Line 6 by itself does not cause a problem, only when it is paired with Line 7. One way to fix this code so it compiles would be to add an `if-then` statement on line 6. The other solution would be to remove line 7.
7. D. As you learned in the section "Ternary Operator," although parentheses are not required, they do greatly increase code readability, such as the following equivalent statement:

   ```
   System.out.println((x > 2) ? ((x < 4) ? 10 : 8) : 7)
   ```

   We apply the outside ternary operator first, as it is possible the inner ternary expression may never be evaluated. Since `(x>2)` is `true`, this reduces the problem to:

   ```
   System.out.println((x < 4) ? 10 : 8)
   ```

   Since `x` is greater than `2`, the answer is `8`, or option D in this case.
8. B. This example is tricky because of the second assignment operator embedded in line 5. The expression `(z=false)` assigns the value `false` to `z` and returns `false` for the entire expression. Since `y` does not equal `10`, the left-hand side returns `true`; therefore, the exclusive or `(^)` of the entire expression assigned to `x` is `true`. The output reflects these assignments, with no change to `y`, so option B is the only correct answer. The code compiles and runs without issue, so option F is not correct.
9. F. In this example, the update statement of the `for` loop is missing, which is fine as the statement is optional, so option D is incorrect. The expression inside the loop increments `i` but then assigns `i` the old value. Therefore, `i` ends the loop with the same value

that it starts with: `0`. The loop will repeat infinitely, outputting the same statement over and over again because `i` remains `0` after every iteration of the loop.

10. D. Line 4 generates a possible loss of precision compiler error. The `cast` operator has the highest precedence, so it is evaluated first, casting `a` to a byte. Then, the addition is evaluated, causing both `a` and `b` to be promoted to `int` values. The value `90` is an `int` and cannot be assigned to the `byte sum` without an explicit cast, so the code does not compile. The code could be corrected with parentheses around `(a + b)`, in which case option C would be the correct answer.

11. A. The `*` and `%` have the same operator precedence, so the expression is evaluated from left-to-right. The result of `5 * 4` is `20`, and `20 % 3` is `2` (`20` divided by `3` is `18`, the remainder is `2`). The output is `2` and option A is the correct answer.

12. D. The variable `x` is an `int` and `s` is a reference to a `String` object. The two data types are incomparable because neither variable can be converted to the other variable's type. The compiler error occurs on line 5 when the comparison is attempted, so the answer is option D.

13. A. The code compiles successfully, so options C and D are incorrect. The value of `b` after line 4 is `false`. However, the `if-then` statement on line 5 contains an assignment, not a comparison. The variable `b` is assigned `true` on line 3, and the assignment operator returns `true`, so line 5 executes and displays `Success`, so the answer is option A.

14. C. The code compiles successfully, so option F is incorrect. On line 5, the pre-increment operator is used, so `c` is incremented to `4` and the new value is returned to the expression. The value of `result` is computed by adding `4` to the original value of `8`, resulting in a new value of `12`, which is output on line 6. Therefore, option C is the correct answer.

15. E. This is actually a much simpler problem than it appears to be. The `while` statement on line 4 is missing parentheses, so the code will not compile, and option E is the correct answer. If the parentheses were added, though, option F would be the correct answer since the loop does not use curly braces to include `x++` and the `boolean` expression never changes. Finally, if curly braces were added around both expressions, the output would be `10, 6` and option B would be correct.

16. D. The variable `y` is declared within the body of the `do-while` statement, so it is out of scope on line 6. Line 6 generates a compiler error, so option D is the correct answer.

17. D. The code compiles without issue, so option F is incorrect. After the first execution of the loop, `i` is decremented to `9` and `result` to `13`. Since `i` is not 8, `keepGoing` is `false`, and the loop continues. On the next iteration, `i` is decremented to `8` and `result` to `11`. On the second execution, `i` does equal `8`, so `keepGoing` is set to `false`. At the conclusion of the loop, the loop terminates since `keepGoing` is no longer `true`. The value of `result` is `11`, and the correct answer is option D.

18. A. The expression on line 5 is `true` when `row * col` is an even number. On the first iteration, `row = 1` and `col = 1`, so the expression on line 6 is `false`, the `continue` is skipped, and `count` is incremented to `1`. On the second iteration, `row = 1` and

`col = 2`, so the expression on line 6 is `true` and the `continue` ends the outer loop with `count` still at `1`. On the third iteration, `row = 2` and `col = 1`, so the expression on line 6 is `true` and the `continue` ends the outer loop with `count` still at `1`. On the fourth iteration, `row = 3` and `col = 1`, so the expression on line 6 is `false`, the `continue` is skipped, and `count` is incremented to `2`. Finally, on the fifth and final iteration, `row = 3` and `col = 2`, so the expression on line 6 is `true` and the `continue` ends the outer loop with `count` still at `2`. The result of `2` is displayed, so the answer is option B.

19. D. Prior to the first iteration, `m = 9`, `n = 1`, and `x = 0`. After the iteration of the first loop, `m` is updated to `8`, `n` to `3`, and `x` to the sum of the new values for `m + n`, `0 + 11 = 11`. After the iteration of the second loop, `m` is updated to `7`, `n` to `5`, and `x` to the sum of the new values for `m + n`, `11 + 12 = 23`. After the iteration of the third loop, `m` is updated to `6`, `n` to `7`, and `x` to the sum of the new values for `m + n`, `23 + 13 = 36`. On the fourth iteration of the loop, `m > n` evaluates to `false`, as `6 < 7` is not `true`. The loop ends and the most recent value of `x`, `36`, is output, so the correct answer is option D.

20. B. The code compiles and runs without issue, so options C, D, and E are not correct. The value of `grade` is `'B'` and there is a matching `case` statement that will cause `"great"` to be printed. There is no `break` statement after the `case`, though, so the next `case` statement will be reached, and `"good"` will be printed. There is a `break` after this `case` statement, though, so the `switch` statement will end. The correct answer is thus option B.

Chapter 3: Core Java APIs

1. G. Line 5 does not compile. This question is checking to see if you are paying attention to the types. `numFish` is an `int` and `1` is an `int`. Therefore, we use numeric addition and get 5. The problem is that we can't store an `int` in a `String` variable. Supposing line 5 said `String anotherFish = numFish + 1 + "";`. In that case, the answer would be options A and D. The variable defined on line 5 would be the string `"5"`, and both output statements would use concatenation.

2. A, C, D. The code compiles fine. Line 3 points to the `String` in the string pool. Line 4 calls the `String` constructor explicitly and is therefore a different object than `s`. Lines 5 and 7 check for object equality, which is true, and so print `one` and `three`. Line 6 uses object reference equality, which is not true since we have different objects. Line 7 also compares references but is true since both references point to the object from the string pool. Finally, line 8 compares one object from the string pool with one that was explicitly constructed and returns `false`.

3. B, C, E. Immutable means the state of an object cannot change once it is created. Immutable objects can be garbage collected just like mutable objects. `String` is immutable. `StringBuilder` can be mutated with methods like `append()`. Although

`StringBuffer` isn't on the exam, you should know about it anyway in case older questions haven't been removed.

4. B. This example uses method chaining. After the call to `append()`, `sb` contains `"aaa"`. That result is passed to the first `insert()` call, which inserts at index 1. At this point `sb` contains `abbbaa`. That result is passed to the final `insert()`, which inserts at index 4, resulting in `abbaccca`.

5. F. The question is trying to distract you into paying attention to logical equality versus object reference equality. It is hoping you will miss the fact that line 4 does not compile. Java does not allow you to compare `String` and `StringBuilder` using `==`.

6. B. A `String` is immutable. Calling `concat()` returns a new `String` but does not change the original. A `StringBuilder` is mutable. Calling `append()` adds characters to the existing character sequence along with returning a reference to the same object.

7. B, D, E. `length()` is simply a count of the number of characters in a `String`. In this case, there are six characters. `charAt()` returns the character at that index. Remember that indexes are zero based, which means that index 3 corresponds to `d` and index 6 corresponds to 1 past the end of the array. A `StringIndexOutOfBoundsException` is thrown for the last line.

8. A, D, E. `substring()` has two forms. The first takes the index to start with and the index to stop immediately before. The second takes just the index to start with and goes to the end of the `String`. Remember that indexes are zero based. The first call starts at index 1 and ends with index 2 since it needs to stop before index 3. The second call starts at index 7 and ends in the same place, resulting in an empty `String`. This prints out a blank line. The final call starts at index 7 and goes to the end of the `String`.

9. C. This question is trying to see if you know that `String` objects are immutable. Line 4 returns `"PURR"` but the result is ignored and not stored in `s`. Line 5 returns `"purr"` since there is no whitespace present but the result is again ignored. Line 6 returns `"ur"` because it starts with index 1 and ends before index 3 using zero-based indexes. The result is ignored again. Finally, on line 6 something happens. We concatenate four new characters to `s` and now have a `String` of length 8.

10. F. `a += 2` expands to `a = a + 2`. A `String` concatenated with any other type gives a `String`. Lines 14, 15, and 16 all append to `a`, giving a result of `"2cfalse"`. The `if` statement on line 18 returns `false` because the values of the two `String` objects are the same using object equality. The `if` statement on line 17 returns `false` because the two `String` objects are not the same in memory. One comes directly from the string pool and the other comes from building using `String` operations.

11. E. Line 6 adds 1 to `total` because `substring()` includes the starting index but not the ending index. Line 7 adds 0 to `total`. Line 8 is a problem: Java does not allow the indexes to be specified in reverse order and the code throws a `StringIndexOutOfBoundsException`.

12. A. First, we delete the characters at index 2 until the character one before index 8. At this point, 0189 is in `numbers`. The following line uses method chaining. It appends a dash to the end of the characters sequence, resulting in `0189-`, and then inserts a plus sign at index 2, resulting in `01+89-`.

13. F. This is a trick question. The first line does not compile because you cannot assign a `String` to a `StringBuilder`. If that line were `StringBuilder b = new StringBuilder("rumble")`, the code would compile and print `rum4`. Watch out for this sort of trick on the exam. You could easily spend a minute working out the character positions for no reason at all.

14. A, C. The `reverse()` method is the easiest way of reversing the characters in a `StringBuilder`; therefore, option A is correct. Option B is a nice distraction—it does in fact return `"avaJ"`. However, `substring()` returns a `String`, which is not stored anywhere. Option C uses method chaining. First it creates the value `"JavavaJ$"`. Then it removes the first three characters, resulting in `"avaJ$"`. Finally, it removes the last character, resulting in `"avaJ"`. Option D throws an exception because you cannot delete the character after the last index. Remember that `deleteCharAt()` uses indexes that are zero based and `length()` counts starting with 1.

15. C, E, F. Option C uses the variable name as if it were a type, which is clearly illegal. Options E and F don't specify any size. Although it is legal to leave out the size for later dimensions of a multidimensional array, the first one is required. Option A declares a legal 2D array. Option B declares a legal 3D array. Option D declares a legal 2D array. Remember that it is normal to see on the exam types you might not have learned. You aren't expected to know anything about them.

16. C. Arrays define a property called `length`. It is not a method, so parentheses are not allowed.

17. F. The `ArrayList` class defines a method called `size()`.

18. A, C, D, E. An array is not able to change size and can have multiple dimensions. Both an array and `ArrayList` are ordered and have indexes. Neither is immutable. The elements can change in value.

19. B, C. An array does not override `equals()` and so uses object equality. `ArrayList` does override `equals()` and defines it as the same elements in the same order. The compiler does not know when an index is out of bounds and thus can't give you a compiler error. The code will throw an exception at runtime, though.

20. D. The code does not compile because `list` is instantiated using generics. Only `String` objects can be added to `list` and 7 is an `int`.

21. C. After line 4, `values` has one element (4). After line 5, `values` has two elements (4, 5). After line 6, `values` has two elements (4, 6) because `set()` does a replace. After line 7, `values` has only one element (6).

22. D. The code compiles and runs fine. However, an array must be sorted for `binarySearch()` to return a meaningful result.

23. A. Line 4 creates a fixed size array of size 4. Line 5 sorts it. Line 6 converts it back to an array. The brackets aren't in the traditional place, but they are still legal. Line 7 prints the first element, which is now –1.

24. C. Converting from an array to an `ArrayList` uses `Arrays.asList(names)`. There is no `asList()` method on an array instance. If this code were corrected to compile, the answer would be option A.

25. D. After sorting, `hex` contains `[30, 3A, 8, FF]`. Remember that numbers sort before letters and strings sort alphabetically. This makes 30 come before 8. A binary search correctly finds 8 at index 2 and 3A at index 1. It cannot find 4F but notices it should be at index 2. The rule when an item isn't found is to negate that index and subtract 1. Therefore, we get –2–1, which is –3.

26. A, B, D. Lines 5 and 7 use autoboxing to convert an `int` to an `Integer`. Line 6 does not because `valueOf()` returns an `Integer`. Line 8 does not because `null` is not an `int`. The code does not compile. However, when the `for` loop tries to unbox `null` into an `int`, it fails and throws a `NullPointerException`.

27. B. The first `if` statement is false because the variables do not point to the same object. The second `if` statement is true because `ArrayList` implements equality to mean the same elements in the same order.

28. D, F. Options A and B are incorrect because `LocalDate` does not have a public constructor. Option C is incorrect because months start counting with 1 rather than 0. Option E is incorrect because it uses the old pre–Java 8 way of counting months, again beginning with 0. Options D and F are both correct ways of specifying the desired date.

29. D. A `LocalDate` does not have a time element. Therefore, it has no method to add hours and the code does not compile.

30. F. Java throws an exception if invalid date values are passed. There is no 40th day in April—or any other month for that matter.

31. B. The date starts out as April 30, 2018. Since dates are immutable and the plus methods have their return values ignored, the result is unchanged. Therefore, option B is correct.

32. E. Even though `d` has both date and time, the formatter only outputs time.

33. B. `Period` does not allow chaining. Only the last `Period` method called counts, so only the two years are subtracted.

Chapter 4: Methods and Encapsulation

1. B, C. `void` is a return type. Only the access modifier or optional specifiers are allowed before the return type. Option C is correct, creating a method with private access. Option B is correct, creating a method with default access and the optional specifier `final`. Since default access does not require a modifier, we get to jump right to `final`.

Option A is incorrect because default access omits the access modifier rather than specifying default. Option D is incorrect because Java is case sensitive. It would have been correct if `public` were the choice. Option E is incorrect because the method already has a `void` return type. Option F is incorrect because labels are not allowed for methods.

2. A, D. Options A and D are correct because the optional specifiers are allowed in any order. Options B and C are incorrect because they each have two return types. Options E and F are incorrect because the return type is before the optional specifier and access modifier, respectively.

3. A, C, D. Options A and C are correct because a `void` method is allowed to have a return statement as long as it doesn't try to return a value. Options B and G do not compile because `null` requires a reference object as the return type. `void` is not a reference object since it is a marker for no return type. `int` is not a reference object since it is a primitive. Option D is correct because it returns an `int` value. Option E does not compile because it tries to return a `double` when the return type is `int`. Since a `double` cannot be assigned to an `int`, it cannot be returned as one either. Option F does not compile because no value is actually returned.

4. A, B, G. Options A and B are correct because the single vararg parameter is the last parameter declared. Option G is correct because it doesn't use any vararg parameters at all. Options C and F are incorrect because the vararg parameter is not last. Option D is incorrect because two vararg parameters are not allowed in the same method. Option E is incorrect because the `...` for a vararg must be after the type, not before it.

5. D, G. Option D passes the initial parameter plus two more to turn into a vararg array of size 2. Option G passes the initial parameter plus an array of size 2. Option A does not compile because it does not pass the initial parameter. Options E and F do not compile because they do not declare an array properly. It should be `new boolean[] {true}`. Option B creates a vararg array of size 0 and option C creates a vararg array of size 1.

6. D. Option D is correct. This is the common implementation for encapsulation by setting all fields to be private and all methods to be public. Option A is incorrect because protected access allows everything that package private access allows and additionally allows subclasses access. Option B is incorrect because the class is public. This means that other classes can see the class. However, they cannot call any of the methods or read any of the fields. It is essentially a useless class. Option C is incorrect because package private access applies to the whole package. Option E is incorrect because Java has no such capability.

7. B, C, D, F. The two classes are in different packages, which means private access and default (package private) access will not compile. Additionally, protected access will not compile since `School` does not inherit from `Classroom`. Therefore, only line 8 will compile because it uses public access.

8. B, C, E. Encapsulation requires using methods to get and set instance variables so other classes are not directly using them. Instance variables must be private for this to work. Immutability takes this a step further, allowing only getters, so the instance variables do not change state.

9. C, E. Option A is incorrect because the property is of type `boolean` and getters must begin with `is` for `boolean`s. Options B and D are incorrect because they don't follow the naming convention of beginning with `get`/`is`/`set`. Options C and E follow normal getter and setter conventions.

10. B. `Rope` runs line 3, setting `LENGTH` to 5, then immediately after runs the `static` initializer, which sets it to 10. Line 5 calls the `static` method normally and prints `swing`. Line 6 also calls the `static` method. Java allows calling a `static` method through an instance variable. Line 7 uses the static import on line 2 to reference `LENGTH`.

11. B, E. Line 10 does not compile because static methods are not allowed to call instance methods. Even though we are calling `play()` as if it were an instance method and an instance exists, Java knows `play()` is really a static method and treats it as such. If line 10 is removed, the code works. It does not throw a `NullPointerException` on line 16 because `play()` is a static method. Java looks at the type of the reference for `rope2` and translates the call to `Rope.play()`.

12. D. There are two details to notice in this code. First, note that `RopeSwing` has an instance initializer and not a `static` initializer. Since `RopeSwing` is never constructed, the instance initializer does not run. The other detail is that *`length`* is `static`. Changes from one object update this common `static` variable.

13. E. `static final` variables must be set exactly once, and it must be in the declaration line or in a `static` initialization block. Line 4 doesn't compile because *`bench`* is not set in either of these locations. Line 15 doesn't compile because `final` variables are not allowed to be set after that point. Line 11 doesn't compile because *`name`* is set twice: once in the declaration and again in the static block. Line 12 doesn't compile because *`rightRope`* is set twice as well. Both are in static initialization blocks.

14. B. The two valid ways to do this are `import static java.util.Collections.*;` and `import static java.util.Collections.sort;`. Option A is incorrect because you can only do a static `import` on `static` members. Classes such as `Collections` require a regular `import`. Option C is nonsense as method parameters have no business in an `import`. Options D, E, and F try to trick you into reversing the syntax of `import static`.

15. E. The argument on line 17 is a `short`. It can be promoted to an `int`, so `print()` on line 5 is invoked. The argument on line 18 is a `boolean`. It can be autoboxed to a `boolean`, so `print()` on line 11 is invoked. The argument on line 19 is a `double`. It can be autoboxed to a `double`, so `print()` on line 11 is invoked. Therefore, the output is `intObjectObject` and the correct answer is option E.

16. B. Since Java is pass-by-value and the variable on line 8 never gets reassigned, it stays as 9. In the method `square`, *`x`* starts as 9. *`y`* becomes 81 and then *`x`* gets set to –1. Line 9 does set *`result`* to 81. However, we are printing out *`value`* and that is still 9.

17. B, D, E. Since Java is pass-by-reference, assigning a new object to *`a`* does not change the caller. Calling `append()` does affect the caller because both the method parameter and

caller have a reference to the same object. Finally, returning a value does pass the reference to the caller for assignment to *s3*.

18. C, G. Since the `main()` method is in the same class, it can call `private` methods in the class. `this()` may only be called as the first line of a constructor. `this.variableName` can be called from any instance method to refer to an instance variable. It cannot be called from a static method because there is no instance of the class to refer to. Option F is tricky. The default constructor is only written by the compiler if no user-defined constructors were provided. `this()` can only be called from a constructor in the same class. Since there can be no user-defined constructors in the class if a default constructor was created, it is impossible for option F to be true.

19. A, G. Options B and C don't compile because the constructor name must match the classname. Since Java is case sensitive, these don't match. Options D, E, and F all compile and provide one user-defined constructor. Since a constructor is coded, a default constructor isn't supplied. Option G defines a method, but not a constructor. Option A does not define a constructor, either. Since no constructor is coded, a default constructor is provided for options A and G.

20. E. Options A and B will not compile because constructors cannot be called without `new`. Options C and D will compile but will create a new object rather than setting the fields in this one. Option F will not compile because `this()` must be the first line of a constructor. Option E is correct.

21. C. Within the constructor `numSpots` refers to the constructor parameter. The instance variable is hidden because they have the same name. `this.numSpots` tells Java to use the instance variable. In the `main()` method, `numSpots` refers to the instance variable. Option A sets the constructor parameter to itself, leaving the instance variable as 0. Option B sets the constructor parameter to the value of the instance variable, making them both 0. Option C is correct, setting the instance variable to the value of the constructor parameter. Options D and E do not compile.

22. E. On line 3 of `OrderDriver`, we refer to `Order` for the first time. At this point the statics in `Order` get initialized. In this case, the statics are the `static` declaration of `result` and the `static` initializer. `result` is `u` at this point. On line 4, `result` is the same because the `static` initialization is only run once. On line 5, we create a new `Order`, which triggers the instance initializers in the order they appear in the file. Now `result` is `ucr`. Line 6 creates another `Order`, triggering another set of initializers. Now `result` is `ucrcr`. Notice how the `static` is on a different line than the initialization code in lines 4–5 of `Order`. The exam may try to trick you by formatting the code like this to confuse you.

23. A. Line 4 instantiates an `Order`. Java runs the declarations and instance initializers first in the order they appear. This sets `value` to `tacf`. Line 5 creates another `Order` and initializes `value` to `tacb`. The object on line 5 is stored in the same variable line 4 used. This makes the object created on line 4 unreachable. When `value` is printed, it is the instance variable in the object created on line 5.

24. B, C, E. *value1* is a `final` instance variable. It can only be set once: in the variable declaration, an instance initializer, or a constructor. Option A does not compile because the `final` variable was already set in the declaration. *value2* is a `static` variable. Both instance and static initializers are able to access `static` variables, making options B and E correct. *value3* is an instance variable. Options D and F do not compile because a `static` initializer does not have access to instance variables.

25. A, E. The `100` parameter is an `int` and so calls the matching `int` constructor. When this constructor is removed, Java looks for the next most specific constructor. Java prefers autoboxing to varargs, and so chooses the `Integer` constructor. The `100L` parameter is a `long`. Since it can't be converted into a smaller type, it is autoboxed into a `Long` and then the constructor for `Object` is called.

26. A. This code is correct. Line 8 creates a lambda expression that checks if the age is less than 5. Since there is only one parameter and it does not specify a type, the parentheses around the type parameter are optional. Line 10 uses the `Predicate` interface, which declares a `test()` method.

27. C. The interface takes two `int` parameters. The code on line 7 attempts to use them as if one is a `StringBuilder`. It is tricky to use types in a lambda when they are implicitly specified. Remember to check the interface for the real type.

28. A, D, F. `removeIf()` expects a `Predicate`, which takes a parameter list of one parameter using the specified type. Options B and C are incorrect because they do not use the `return` keyword. It is required inside braces for lambda bodies. Option E is incorrect because it is missing the parentheses around the parameter list. This is only optional for a single parameter with an inferred type.

29. A, F. Option B is incorrect because it does not use the `return` keyword. Options C, D, and E are incorrect because the variable *e* is already in use from the lambda and cannot be redefined. Additionally, option C is missing the `return` keyword and option E is missing the semicolon.

Chapter 5: Class Design

1. B. All interface methods are implicitly `public`, so option B is correct and option A is not. Interface methods may be declared as `static` or `default` but are never implicitly added, so options C and F are incorrect. Option D is incorrect—`void` is not a modifier; it is a return type. Option E is a tricky one, because prior to Java 8 all interface methods would be assumed to be `abstract`. Since Java 8 now includes default and static methods and they are never abstract, you cannot assume the `abstract` modifier will be implicitly applied to all methods by the compiler.

2. E. The code will not compile because the parent class `Mammal` doesn't define a no-argument constructor, so the first line of a `Platypus` constructor should be an explicit call to `super(int age)`. If there was such a call, then the output would be `MammalPlatypus`, since the super constructor is executed before the child constructor.

3. A, B, D, E. The blank can be filled with any class or interface that is a supertype of `TurtleFrog`. Option A is a superclass of `TurtleFrog`, and option B is the same class, so both are correct. `BrazilianHornedFrog` is not a superclass of `TurtleFrog`, so option C is incorrect. `TurtleFrog` inherits the `CanHope` interface, so option D is correct. All classes inherit `Object`, so option E is correct. Finally, `Long` is an unrelated class that is not a superclass of `TurtleFrog`, and is therefore incorrect.
4. C, E. The code doesn't compile, so option A is incorrect. Option B is also not correct because the rules for overriding a method allow a subclass to define a method with an exception that is a subclass of the exception in the parent method. Option C is correct because the return types are not covariant; in particular, `Number` is not a subclass of `Integer`. Option D is incorrect because the subclass defines a method that is more accessible than the method in the parent class, which is allowed. Finally, option E is correct because the method is declared as `static` in the parent class and not so in the child class. For nonprivate methods in the parent class, both methods must use `static` (hide) or neither should use `static` (override).
5. A, D, E, F. First off, options B and C are incorrect because protected and public methods may be overridden, not hidden. Option A is correct because private methods are always hidden in a subclass. Option D is also correct because static methods cannot be overridden, only hidden. Options E and F are correct because variables may only be hidden, regardless of the access modifier.
6. D. The code fails to compile because `Beetle`, the first concrete subclass, doesn't implement `getNumberOfSections()`, which is inherited as an abstract method; therefore, option D is correct. Option B is incorrect because there is nothing wrong with this interface method definition. Option C is incorrect because an abstract class is not required to implement any abstract methods, including those inherited from an interface. Option E is incorrect because the code fails at compilation-time.
7. B, C. A reference to an object requires an explicit cast if referenced with a subclass, so option A is incorrect. If the cast is to a superclass reference, then an explicit cast is not required. Because of polymorphic parameters, if a method takes the superclass of an object as a parameter, then any subclass references may be used without a cast, so option B is correct. All objects extend `java.lang.Object`, so if a method takes that type, any valid object, including `null`, may be passed; therefore, option C is correct. Some cast exceptions can be detected as errors at compile-time, but others can only be detected at runtime, so D is incorrect. Due to the nature of polymorphism, a `public` instance method can be overridden in a subclass and calls to it will be replaced even in the superclass it was defined, so E is incorrect.
8. F. The interface variable amount is correctly declared, with `public` and `static` being assumed and automatically inserted by the compiler, so option B is incorrect. The method declaration for `eatGrass()` on line 3 is incorrect because the method has been marked as `static` but no method body has been provided. The method declaration for `chew()` on line 4 is also incorrect, since an interface method that provides a body must be marked as `default` or `static` explicitly. Therefore, option F is the correct answer since this code contains two compile-time errors.

9. A. Although the definition of methods on lines 2 and 5 vary, both will be converted to `public abstract` by the compiler. Line 4 is fine, because an interface can have `public` or default access. Finally, the class `Falcon` doesn't need to implement the interface methods because it is marked as `abstract`. Therefore, the code will compile without issue.

10. B, C, E, F. Option A is wrong, because an abstract class may contain concrete methods. Since Java 8, interfaces may also contain concrete methods in form of static or default methods. Although all variables in interfaces are assumed to be `public static final`, abstract classes may contain them as well, so option B is correct. Both abstract classes and interfaces can be extended with the `extends` keyword, so option C is correct. Only interfaces can contain default methods, so option D is incorrect. Both abstract classes and interfaces can contain static methods, so option E is correct. Both structures require a concrete subclass to be instantiated, so option F is correct. Finally, though an instance of an object that implements an interface inherits `java.lang.Object`, the interface itself doesn't; otherwise, Java would support multiple inheritance for objects, which it doesn't. Therefore, option G is incorrect.

11. A, D, E. Interface variables are assumed to be `public static final`; therefore, options A, D, and E are correct. Options B and C are incorrect because interface variables must be public—interfaces are implemented by classes, not inherited by interfaces. Option F is incorrect because variables can never be abstract.

12. B. This code compiles and runs without issue, outputting `false`, so option B is the correct answer. The first declaration of `isBlind()` is as a default interface method, assumed `public`. The second declaration of `isBlind()` correctly overrides the default interface method. Finally, the newly created `Owl` instance may be automatically cast to a `Nocturnal` reference without an explicit cast, although adding it doesn't break the code.

13. A. The code compiles and runs without issue, so options E and F are incorrect. The `printName()` method is an overload in `Spider`, not an override, so both methods may be called. The call on line 8 references the version that takes an `int` as input defined in the `Spider` class, and the call on line 9 references the version in the `Arthropod` class that takes a `double`. Therefore, `SpiderArthropod` is output and option A is the correct answer.

14. C. The code compiles without issue, so option A is wrong. Option B is incorrect, since an abstract class could implement `HasVocalCords` without the need to override the `makeSound()` method. Option C is correct; any class that implements `CanBark` automatically inherits its methods, as well as any inherited methods defined in the parent interface. Because option C is correct, it follows that option D is incorrect. Finally, an interface can extend multiple interfaces, so option E is incorrect.

15. B. Concrete classes are, by definition, not abstract, so option A is incorrect. A concrete class must implement all inherited abstract methods, so option B is correct. Option C is incorrect; a superclass may have already implemented an inherited interface, so the concrete subclass would not need to implement the method. Concrete classes can be both `final` and not `final`, so option D is incorrect. Finally, abstract methods must be overridden by a concrete subclass, so option E is incorrect.

16. E. The code doesn't compile, so options A and B are incorrect. The issue with line 9 is that `layEggs()` is marked as `final` in the superclass `Reptile`, which means it cannot be overridden. There are no errors on any other lines, so options C and D are incorrect.
17. B. This may look like a complex question, but it is actually quite easy. Line 2 contains an invalid definition of an abstract method. Abstract methods cannot contain a body, so the code will not compile and option B is the correct answer. If the body `{}` was removed from line 2, the code would still not compile, although it would be line 8 that would throw the compilation error. Since `dive()` in `Whale` is abstract and `Orca` extends `Whale`, then it must implement an overridden version of `dive()`. The method on line 9 is an overloaded version of `dive()`, not an overridden version, so `Orca` is an invalid subclass and will not compile.
18. E. The code doesn't compile because line 6 contains an incompatible override of the `getNumberOfGills(int input)` method defined in the `Aquatic` interface. In particular, `int` and `String` are not covariant returns types, since `int` is not a subclass of `String`. Note that line 5 compiles without issue; `getNumberOfGills()` is an overloaded method that is not related to the parent interface method that takes an `int` value.
19. A, C, F. First off, `Cobra` is a subclass of `Snake`, so option A can be used. `GardenSnake` is not defined as a subclass of `Snake`, so it cannot be used and option B is incorrect. The class `Snake` is not marked as `abstract`, so it can be instantiated and passed, so option C is correct. Next, `Object` is a superclass of `Snake`, not a subclass, so it also cannot be used and option D is incorrect. The class `String` is unrelated in this example, so option E is incorrect. Finally, a `null` value can always be passed as an object value, regardless of type, so option F is correct.
20. A. The code compiles and runs without issue, so options C, D, and E are incorrect. The trick here is that the method `fly()` is marked as `private` in the parent class `Bird`, which means it may only be hidden, not overridden. With hidden methods, the specific method used depends on where it is referenced. Since it is referenced within the `Bird` class, the method declared on line 2 was used, and option A is correct. Alternatively, if the method was referenced within the `Pelican` class, or if the method in the parent class was marked as `protected` and overridden in the subclass, then the method on line 9 would have been used.

Chapter 6: Exceptions

1. B. Runtime exceptions are also known as unchecked exceptions. They are allowed to be declared, but they don't have to be. Checked exceptions must be handled or declared. Legally, you can handle `java.lang.Error` subclasses, but it's not a good idea.
2. B, D. In a method declaration, the keyword `throws` is used. To actually throw an exception, the keyword `throw` is used and a new exception is created.
3. C. A `try` statement is required to have a `catch` clause and/or `finally` clause. If it goes the `catch` route, it is allowed to have multiple `catch` clauses.

4. B. The second line tries to cast an `Integer` to a `String`. Since `String` does not extend `Integer`, this is not allowed and a `ClassCastException` is thrown.
5. A, B, D. `java.io.IOException` is thrown by many methods in the `java.io` package, but it is always thrown programmatically. The same is true for `NumberFormatException`; it is thrown programmatically by the wrapper classes of `java.lang`. The other three exceptions are all thrown by the JVM when the corresponding problem arises.
6. C. The compiler tests the operation for a valid type but not a valid result, so the code will still compile and run. At runtime, evaluation of the parameter takes place before passing it to the `print()` method, so an `ArithmeticException` object is raised.
7. C. The `main()` method invokes `go` and `A` is printed on line 3. The `stop` method is invoked and `E` is printed on line 14. Line 16 throws a `NullPointerException`, so `stop` immediately ends and line 17 doesn't execute. The exception isn't caught in `go`, so the `go` method ends as well, but not before its `finally` block executes and `C` is printed on line 9. Because `main()` doesn't catch the exception, the stack trace displays and no further output occurs, so `AEC` was the output printed before the stack trace.
8. E. The order of `catch` blocks is important because they're checked in the order they appear after the `try` block. Because `ArithmeticException` is a child class of `RuntimeException`, the `catch` block on line 7 is unreachable. (If an `ArithmeticException` is thrown in `try` try block, it will be caught on line 5.) Line 7 generates a compiler error because it is unreachable code.
9. B. The `main()` method invokes `start` on a new `Laptop` object. Line 4 prints `Starting up`; then line 5 throws an `Exception`. Line 6 catches the exception, line 7 prints `Problem`, and then line 8 calls `System.exit`, which terminates the JVM. The `finally` block does not execute because the JVM is no longer running.
10. E. The `parseName` method is invoked within `main()` on a new `Dog` object. Line 4 prints `1`. The `try` block executes and `2` is printed. Line 7 throws a `NumberFormatException`, so line 8 doesn't execute. The exception is caught on line 9, and line 10 prints `4`. Because the exception is handled, execution resumes normally. `parseName` runs to completion, and line 17 executes, printing `5`. That's the end of the program, so the output is `1245`.
11. A. The `parseName` method is invoked on a new `Cat` object. Line 4 prints `1`. The `try` block is entered, and line 6 prints `2`. Line 7 throws a `NumberFormatException`. It isn't caught, so `parseName` ends. `main()` doesn't catch the exception either, so the program terminates and the stack trace for the `NumberFormatException` is printed.
12. A, B, D, G. The `main()` method invokes run on a new `Mouse` object. Line 4 prints `1` and line 6 prints `2`, so options A and B are correct. Line 7 throws a `NullPointerException`, which causes line 8 to be skipped, so C is incorrect. The exception is caught on line 9 and line 10 prints `4`, so option D is correct. Line 11 throws the exception again, which causes `run()` to immediately end, so line 13 doesn't execute and option E is incorrect. The `main()` method doesn't catch the exception either, so line 18 doesn't execute and option F is incorrect. The uncaught `NullPointerException` causes the stack trace to be printed, so option G is correct.

13. A, B, C, E. Classes listed in the `throws` part of a method declaration must extend `java.lang.Throwable`. This includes `Error`, `Exception`, and `RuntimeException`. Arbitrary classes such as `String` can't go there. Any Java type, including `Exception`, can be declared as the return type. However, this will simply return the object rather than throw an exception.

14. A, C, D, E. A method that declares an exception isn't required to throw one, making option A correct. Runtime exceptions can be thrown in any method, making options C and E correct. Option D matches the exception type declared and so is also correct. Option B is incorrect because a broader exception is not allowed.

15. A, B, D, E. `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, and `NumberFormatException` are runtime exceptions. Sorry, you have to memorize them. Any class that extends `RuntimeException` is a runtime (unchecked) exception. Classes that extend `Exception` but not `RuntimeException` are checked exceptions.

16. B. `IllegalArgumentException` is used when an unexpected parameter is passed into a method. Option A is incorrect because returning `null` or `-1` is a common return value for this scenario. Option D is incorrect because a `for` loop is typically used for this scenario. Option E is incorrect because you should find out how to code the method and not leave it for the unsuspecting programmer who calls your method. Option C is incorrect because you should run!

17. A, C, D, E. The method is allowed to throw no exceptions at all, making option A correct. It is also allowed to throw runtime exceptions, making options D and E correct. Option C is also correct since it matches the signature in the interface.

18. A, B, C, E. Checked exceptions are required to be handled or declared. Runtime exceptions are allowed to be handled or declared. Errors are allowed to be handled or declared, but this is bad practice.

19. C, E. Option C is allowed because it is a more specific type than `RuntimeException`. Option E is allowed because it isn't in the same inheritance tree as `RuntimeException`. It's not a good idea to catch either of these. Option B is not allowed because the method called inside the `try` block doesn't declare an `IOException` to be thrown. The compiler realizes that `IOException` would be an unreachable `catch` block. Option D is not allowed because the same exception can't be specified in two different `catch` blocks. Finally, option A is not allowed because it's more general than `RuntimeException` and would make that block unreachable.

20. A, E. The code begins normally and prints `a` on line 13, followed by `b` on line 15. On line 16, it throws an exception that's caught on line 17. Remember, only the most specific matching `catch` is run. Line 18 prints `c`, and then line 19 throws another exception. Regardless, the `finally` block runs, printing `e`. Since the `finally` block also throws an exception, that's the one printed.